系统基准犯罪条例整理

Rong Lonely

2022 年 4 月 14 日

1 Benchmarking Crimes（基准犯罪）

在审查系统论文时，经常遇到对基准的高度误导性使用。并不是说作者有意误导读者，这同样可能是无能。但这不是借口。

此类情况被称为基准犯罪。倒不是你因为它们进监狱（但可能会?），而是因为它们破坏了科学过程的完整性。而你的论文触犯了其中之一，那么你已经大概率进入被拒行列。其余的工作必须非常好才能被原谅基准犯罪（即使这样，你也会被要求在最终版本中修复它）。

以下是关于基准犯罪的列表，并会持续进行完善。

1.1 Selective benchmarking（选择性基准）

这是所有基准犯罪的根源：使用一组有偏见的基准来（似乎）证明一个观点，这可能与更广泛的评估空间覆盖相矛盾。这充其量是严重无能的明显迹象，或者在最坏的情况下是积极的欺骗企图。

这种犯罪有几种变体，下面列出了最突出的几种。显然，并非所有这种情况都同样糟糕，在某些情况下，这可能只是彻底程度的问题，但以最明目张胆的形式，这是一种真正可怕的罪行。

1.1.1 不评估潜在的性能下降

对于应该提高性能的技术/设计/实现的公平评估实际上必须证明两件事：

- **渐进标准：**在感兴趣的领域，绩效确实有显著提高
- **保守标准：**性能在其他地方没有显著下降

两者都很重要！如果你一方面提高性能，另一方面降低性能，你不能轻易争论说你已经获得了一些东西。

现实情况是，提高性能的技术通常需要一定程度的额外工作：额外的簿记、缓存等。这些东西总是有代价的，假装忽略它是不诚实的。这确实是系统的核心：一切都是为了选择正确的权衡。因此，一项新技术几乎总是会引入一些开销，你需要证明它们是可以接受的。

如果你的创新确实导致了一定程度的退化，那么你需要对其进行分析，并建立一个考虑到其他好处的情况下它是可以接受的案例。但是，如果你只评估你的方法有益的场景，那么你就是在欺骗。没有如果，没有但是。

1.1.2 没有充分理由的基准子设置

经常在 SPEC 基准中看到这种变体（实际上可以是前一个变体的一个实例）。这些套件被设计为套件是有原因的：代表广泛的工作负载，并强调系统的各个方面。

但是，通常不可能在实验系统上运行所有的 SPEC 也是不争的事实。一些 SPEC 程序需要大内存（它们旨在对内存子系统施加压力！）并且可能根本不能在特定平台上运行它们，尤其是嵌入式系统。其他是 FORTRAN 程序，编译器可能不可用。

在这种情况下，不可避免地要选择套件的一个子集。但是，必须清楚地了解结果的价值有限。特别是，如果使用子集，引用 SPEC 的整体品质因数（例如平均加速比）是完全不可接受的！

如果使用子集，则必须充分证明其合理性。每个缺失的程序都必须有令人信服的解释。并且讨论必须小心，不要过多地阅读结果，请记住，可以想象的是，使用的子集观察到的任何趋势都可能被不在子集中的程序恢复。

如果违反了上述规则，读者必然会怀疑作者试图隐瞒某些事情。审稿人对诸如“我们选择了一个有代表性的子集”或“典型结果被显示”之类的表述特别过敏。SPEC 不存在“代表性”子集，“典型”结果很可能是经过精心挑选的，看起来最有利的。对于这样的罪行，不要指望宽恕！

Lmbench 有点特殊。它的许可证实际上禁止报告部分结果，但是完整的 lmbench 运行会产生如此多的结果，以至于不可能在会议论文中报告。另一方面，由于这是探索操作系统各个方面的微基准的集合，因此人们通常了解每个基准测量的内容，并且可能出于充分的理由只对子集感兴趣。在这种情况下，运行特

定的 lmbench 测试具有以明确定义的标准化方式测量特定系统方面的优势。这可能没问题，只要不过多地阅读结果（并且 Larry McVoy 不会因为违反许可而起诉你.....）

这种犯罪的一个变种是**从一个庞大的集合中任意挑选基准**。例如，在描述调试或优化 Linux 驱动程序的方法时，显然有成千上万的候选者。全部使用它们可能是不可行的，你必须选择一个子集。但是，审稿人想了解你为什么选择特定子集。请注意，任意与随机不同，因此随机选择是可以。但是，如果你的选择包含许多晦涩或过时的设备，或者严重偏向于串行和 LED 驱动器，那么审稿人就会怀疑你有什么要隐藏的。

1.1.3 选择性数据集隐藏缺陷

该变体可以再次被视为第一个变体的示例。这里选择输入参数的范围以使系统看起来不错，但该范围并不代表实际工作负载。例如，下面的图表显示了作为负载函数的吞吐量的相当好的可扩展性，并且没有任何进一步的细节，这看起来是一个不错的结果。

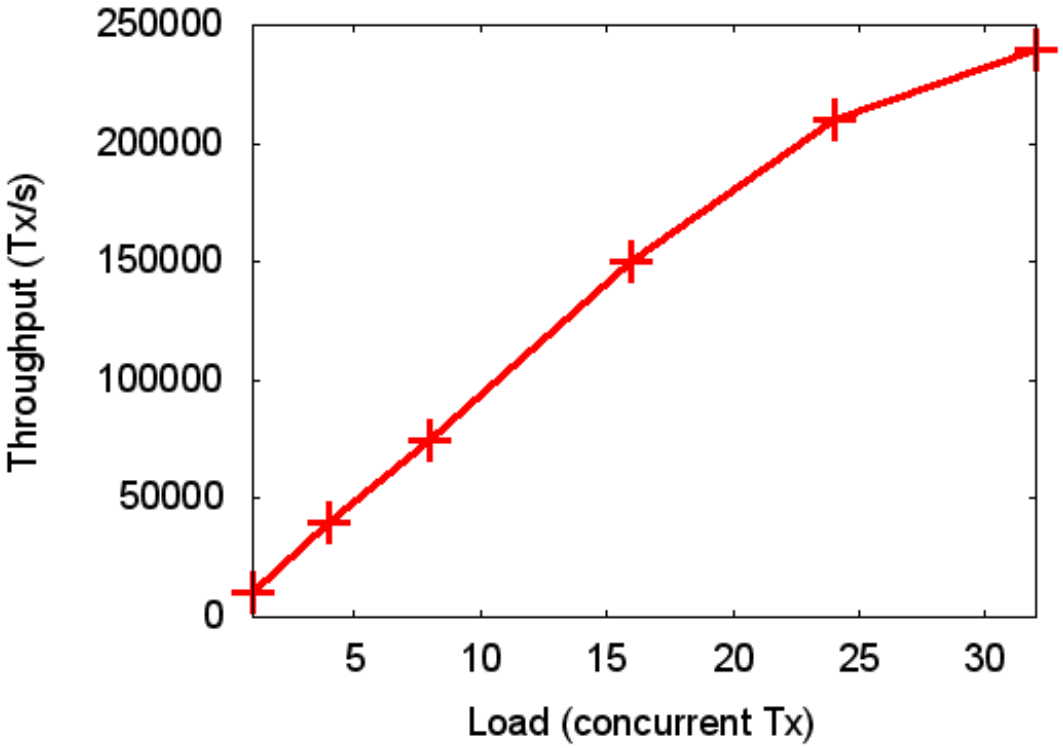

图 1: Throughput

当我们将图表放到上下文中时，情况看起来有点不同。假设这是显示具有不

同数量客户端的数据库系统的吞吐量（每秒事务数）。到现在为止还挺好。

如果我告诉你这是在 32 核机器上测得的，那还那么好吗？然后我们看到的是，只要每个核心最多有一个客户端，吞吐量几乎是线性扩展的。现在这并不是数据库的典型负载。单个事务通常不足以保持一个核心忙碌。为了充分利用你的硬件，你需要运行数据库，以便每个核心平均有多个客户端。

所以，有趣的数据范围从图表结束的地方开始！如果我们将负载增加到真正有趣的范围会发生什么，如下图所示。显然，事情不再看起来那么美好，其实可扩展性依然令人震惊！请注意，虽然有些抽象和简化，但这不是虚构的示例，它

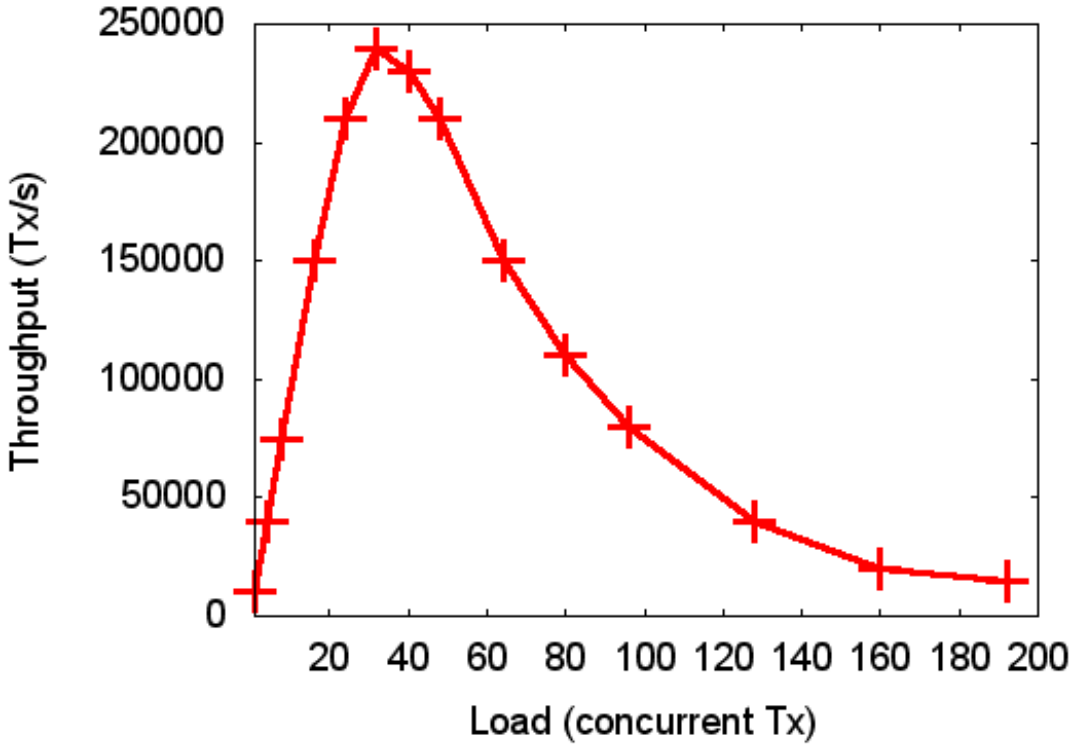

图 2: Throughput

取自真实系统，第一张图与真实出版物中的图等价。第二张图本质上是在同一系统上独立测量的。正如他们所说，根据一个真实的故事...

1.2 基准结果处理不当

1.2.1 假装微基准代表整体性能

微基准专门探测系统的特定方面。即使它们非常全面，也不能代表整个系统的性能。必须使用宏观基准（代表现实世界的工作负载）来提供整体性能的真实

画面。

在极少数情况下，有一个特定的操作被普遍认为是关键的，并且显著的改进被合理地视为实际进展的指示。一个例子是微内核 IPC，它长期以来被认为是一个瓶颈，因此将成本降低一个数量级可能是一个重要的结果。对于一个新的微内核，显示它与已发布的最佳 IPC 性能相匹配可以表明它具有竞争力。

此类例外情况很少见，在大多数情况下，仅基于微基准对系统性能进行争论是不可接受的。

1.2.2 吞吐量下降 x % => 开销为 x %

在审查的论文中可能有 10% 犯了这种恶性罪行。如果系统的吞吐量下降了某个百分比，则根本不能说明增加了相同的百分比开销。恰恰相反，在许多情况下，开销要高得多。为什么?

假设你有一个网络堆栈，它在某些情况下可以达到一定的吞吐量，而修改后的网络堆栈可以减少 10% 的吞吐量。修改引入的开销是多少?

没有进一步的信息，就不可能回答这个问题。为什么吞吐量会下降？为了回答这个问题，我们首先需要了解是什么决定了吞吐量。假设有足够多的传入数据要处理，堆栈可以处理的数据量主要取决于两个因素：处理 (CPU) 成本和延迟。

对实现（不是协议!）的更改将影响处理成本和延迟，但它们对吞吐量的影响是完全不同的。只要 CPU 周期可用，处理成本对吞吐量的影响应该可以忽略不计，而延迟可能会有影响（如果处理速度不够快，数据包将被丢弃）。另一方面，如果 CPU 满载，增加的处理成本将直接转化为延迟。

网络实际上是为了容忍相当多的延迟而设计的，所以它们不应该对它非常敏感。那么，当吞吐量下降时会发生什么?

答案是延迟显著增长以显示吞吐量降低（可能远远超过观察到的吞吐量下降），或者 CPU 已达到极限。如果延迟加倍导致吞吐量下降 10%，那么称其为“10% 开销”可能不太诚实，不是吗?

如果吞吐量最初受到 CPU 功率（满载处理器）的限制，那么吞吐量下降 10% 可以合理地解释为 CPU 成本增加 10%，这可以公平地称为“10% 开销”。但是，如果在原始系统上 CPU 的负载为 60%，而在修改后的系统上却达到 100%（这会导致性能下降），这将怎么办？仍然是“10% 的开销”吗?

显然不是。在这种情况下计算开销的一种公平方法是查看每比特的处理成本，它与 CPU 负载除以吞吐量成正比。按照这个标准，成本上升了 85%。因此，

应将其称为 85% 的开销！

这种情况的一个变体是在“免费”核心上卸载一些处理，而不是将额外核心上的负载包括在处理成本中。那只是作弊。

底线是，提供的信息不完整，这使我们无法真正评估开销/成本，并导致严重低估。吞吐量比较必须始终伴随着完整 CPU 负载的比较。对于 I/O 吞吐量，比较的正确方法是每比特的处理时间！

1.2.3 淡化开销

人们有几种方法可以使他们的开销看起来比实际要小。

- 6% → 13% 的开销增加了 7%

  这个混淆了百分比和百分点，媒体经常（出于无能）这样做。这并不能成为在技术出版物中做同样事情的借口。

  因此，作者修改后的系统将处理开销从 6%（对于原始系统）增加到 13%（对于他们的系统），并且他们羞怯地声称他们只增加了 7% 的开销。当然，那是完全的胡说八道！他们的开销增加了一倍多，他们的系统还不到原来的一半！

  同样，如果你的基准系统的 CPU 利用率为 26%，而你的更改导致利用率为 46%，那么你的负载没有增加 20%，你几乎翻了一番！如果你考虑如果在一台只有一半性能的机器上运行相同的实验会发生什么情况，那么 20% 声明中的不诚实就变得很明显了：负载将从 52% 上升到 92%，显然不是增加 20%！

- 参考点不正确

  这是一种非常常见的相对开销作弊方法：作者根据他们的目的选择分母。例如，基线延迟为 60 秒，作者改进的系统将其减少到 45 秒。然后作者声称“原始系统慢了 33%”（60/45-1 = 0.33）。或者，作者的（以某种方式改进，例如更安全）的系统遭受了一些性能下降，将执行延迟延长到 80 秒，作者却声称“性能仅下降 25%”（1-60/80 = 0.25）。

  这显然是不诚实的。原始系统是基线，因此在计算相对性能时必须出现在分母中。在第一种情况下，正确的值是 1-45/60 = 25% 的改进，而在第二种情况下，它是 80/60-1 = 33% 的退化。

- 其他创造性的开销计算

发表在著名的会议 Usenix ATC 上的论文中有一个不正确计算开销的特别明显的例子。在表 3 中，stat 系统调用的延迟从 0.39μs 上升到 2.28μs，几乎增加了 6 倍。然而，作者称其为“82.89% 的减速”！（还要注意伪准确性；这不是犯罪，而是对数字理解不正确的迹象。）

值得称赞的是，该论文的作者认识到了这个错误并提交了一份勘误表，它更正了开销数据。尽管如此，令人震惊的是，这超出了审阅者的范围。

1.2.4 没有表明数据的重要性

没有任何方差迹象的原始平均值可能具有高度误导性，因为没有迹象表明结果的重要性。来自不同系统的结果之间的任何差异可能只是随机的。

为了表明重要性，必须至少引用标准偏差。系统通常以高度确定性的方式运行，在这种情况下，重复测量的标准偏差可能非常小。在这种情况下，声明“所有标准偏差均低于 1%”可能就足够了。在这种情况下，如果我们看到的效果是 10%，那么读者可以对结果的重要性感到满意。

如果有疑问，请使用学生 t 检验来检查显著性。

此外，如果你将一条线拟合到数据中，请至少引用一个回归系数（除非很明显有很多点并且这条线直接穿过所有这些点）。

1.2.5 用算术平均值平均跨基准分数

算术平均值通常不适用于从一组不同的基准中得出总分（除非各种基准的绝对执行时间具有实际意义）。特别是，如果个别基准分数被归一化（例如针对基线系统），算术平均值就没有意义。

平均的正确方法（即得出一个单一的品质因数）是使用分数的几何平均值

1.3 使用错误的基准

1.3.1 简化模拟系统的基准

有时不可避免地要基于模拟系统进行评估。然而，这是极其危险的，因为模拟始终是一个模型，并且包含一组假设。

因此，必须确保仿真模型不会做出任何会影响你正在寻找的性能方面的简化假设。而且，同样重要的是要让读者/审阅者清楚地知道，你确实已确保该模

型对你的基准具有真正的代表性。

很难就如何做到这一点给出一般性建议。最好的建议是设身处地为读者着想，最好让一个局外人来阅读你的论文并检查你是否真的提出了令人信服的案例。

1.3.2 不恰当和误导性的基准

人们使用应该证明一点的基准，而实际上他们几乎什么也没说（他们唯一可能展示的就是真正糟糕的性能）。例子：

- 使用单处理器基准来衡量多处理器可扩展性

  这似乎完全幼稚，但这并不意味着你在成年人提交的论文中看不到它。有人试图通过运行多个 SPEC CPU 基准副本来展示其系统的多处理器可扩展性。

  当然，这些是不通信的单处理器程序。此外，它们执行的系统调用很少，因此不考验操作系统或底层通信基础设施。它们应该完美地扩展（至少对于低处理器数量）。如果不是，则操作系统或硬件存在严重损坏。真正的可扩展性测试将运行实际跨处理器通信并使用系统调用的工作负载

- 使用 CPU 密集型基准来显示网络开销

  同样，这看起来很愚蠢（或者更确切地说，就是愚蠢），但仍然看到了它。人们试图通过测量 CPU 密集型基准的性能下降来证明他们对 NIC 驱动程序或网络堆栈的更改对性能的影响很小。同样，这可能证明的唯一事情是性能 sux，即是否有任何降级！

1.3.3 用于校准和验证的相同数据集

这是一种相当普遍的犯罪行为。

系统工作经常使用必须针对操作条件（例如平台、工作负载等）进行校准的模型。这是通过一些校准工作负载完成的。然后对系统进行评估，运行评估工作负载，以显示模型的准确性。

不言而喻，校准和评估工作量必须不同！但显然不是这样的。事实上，它们一定是完全不相交的。令人难以置信的是，有多少作者公然违反了这个简单的规则。

当然，使用相同数据进行校准和验证的结果很可能使模型看起来准确，毕竟它是为拟合实验结果而设计的。但所有这样的实验都可以表明模型与现有数据的拟合程度。这并不意味着模型的预测能力，但是对未来测量的预测就是模型的全部内容！

1.4 基准结果比较不当

1.4.1 没有合适的基线

本罪与上述有关。一个典型的案例是通过仅显示两个虚拟化系统的性能来比较不同的虚拟化方法，而没有显示真正的基线案例，那个显然是原生系统。是与原生的比较来决定什么是好是坏，而不是与任意的虚拟化解决方案进行比较！

仔细考虑基线。通常它是最先进的解决方案。通常它是最佳（或理论上最好的）解决方案或硬件限制（假设软件开销为零）。最佳解决方案通常不可能在系统中实现，因为它需要了解未来或神奇的零成本软件，但它通常可以在系统“外部”计算，并且是一个很好的比较基础。在其他情况下，正确的基线在某种意义上是一个不受干扰的系统（如上面的虚拟化示例）。

1.4.2 只对自己进行评估

这是上述犯罪的一种变体，但这并不罕见。与去年的论文相比，你提高了系统的性能可能会让你感到兴奋，但审阅者觉得这不那么令人兴奋。审阅者想看看它的重要性，这意味着与一些公认的标准进行比较。

至少这种罪行的危害性比其他罪行要小，因为它非常明显，而且审阅者很少会上当。

这种犯罪有一个更微妙的变体：根据自身评估模型。有人建立了一个系统模型，做了一些简化的假设，但并非所有假设都明显有效。他们为该问题构建了一个解决方案，然后在包含完全相同假设的模拟系统上评估该解决方案。当然，结果看起来不错，但它们也完全没有价值，因为它们缺乏最基本的现实检查。在已经发表的论文中也发现了很多这种情况。令人沮丧...

1.4.3 对竞争对手进行不公平的基准测试

自己对竞争对手进行基准很棘手，你必须竭尽全力确保不会不公平地对待他们。相信你已经尽可能地调整了你的系统，但是你真的为替代方案付出了同样

的努力吗?

为了让读者/审阅者相信你是公平的，请清楚地描述你对竞争对手系统所做的事情，例如完整描述所有配置参数等。如果你的结果与任何已发布的有关竞争对手系统的数据不匹配，请特别小心. 如果有疑问，请联系该系统的作者以确认你的测量是公平的。

再一次，在一篇发表的论文中看到了这种基准滥用的案例，在那种情况下，“竞争对手”系统是 Gernot 的。这篇论文的作者没有提供任何关于他们如何运行 Gernot 的系统的数据，Gernot 强烈怀疑他们弄错了。例如，Gernot 的开源版本的默认配置已启用调试。关闭该选项（当然，在任何生产环境和任何严肃的性能评估中都会这样做）可以大大提高性能。

底线是，在对竞争对手系统进行自己的基准测试时必须格外小心。很容易以次优方式运行别人的系统，使用次优结果作为比较的基础是非常不道德的，很可能构成学术不端行为。在这种情况下，马虎不是借口!

1.5 丢失的信息

1.5.1 缺少评估平台的规范说明

为了实现可重复性，必须对评估平台进行明确的详细说明，包括可能影响结果的所有特征。平台包括硬件和软件。

细节在很大程度上取决于评估的内容，但至少希望看到处理器架构、内核数量和时钟频率以及内存大小。对于涉及网络的基准，NIC 和交换机支持的吞吐量（如果有的话）。对于运行内存系统的基准，指定所有级别 cache 的大小和相联度通常很重要。一般来说，最好列出 CPU 的型号、内核类型和微架构。

软件也是如此。指定你正在运行的操作系统和（如果使用）管理程序，包括发行版本号。编译器版本通常也是相关的，其他工具的版本也可能是相关的。

1.5.2 缺少子基准结果

在运行基准套件（例如 SPEC）时，通常仅引用该套件的整体品质因数（figure of merit）是不够的。相反，必须显示各个子基准的性能。套件旨在涵盖一系列负载条件，其中一些可能会从你的工作中受益，而另一些则被降低了。只提供总分，最坏情况下会隐藏问题，最好情况是减少可以从评估中获得的见解。

1.5.3 仅提供相对数字

始终给出完整的结果，而不仅仅是比率（除非分母是标准数字）。充其量，只看到相对数字让审阅者怀疑这些数字是否有意义，审阅者被剥夺了一种执行健全性检查的简单方法。在最坏的情况下，它可以掩盖结果真的很糟糕，或者真的不相关。

见过的关于这种犯罪的最糟糕的例子之一是在实际已发表的论文中。它通过显示开销比率：两个相对差异的比率来比较两个系统的性能。从数字中读出任何东西都太相对了。

例如，假设一个系统的开销是另一个系统的两倍。就其本身而言，这告诉我们的很少。也许我们正在比较十倍和二十倍的开销。如果是这样，谁在乎？两者很可能都无法使用。或者一个系统的开销可能是 0.1%，谁在乎另一个系统是否有 0.2% 的开销？底线是我们不知道结果有多重要，但这个表述却意味着它有很大意义。

2 基准测试最佳实践

下面的基准测试规则有点面向操作系统，但基本原则普遍适用

2.1 一般规则

- 确保在开始实验时，系统真正处于静止状态，留出足够的时间以确保所有以前的数据被清除掉。
- 使你的基准测试平台成为我们回归测试套件的一部分。
- 记录你正在做的事情。

2.2 测试数据和结果

始终验证你正在传输的数据。当向磁盘或网络写东西时，读回来并与你写的东西进行比较。读取时，检查你所读取的内容是否正确。

在有些情况下，这将不合理地延长基准测试的时间。如果是这种情况，那么在继续之前，请确保你至少检查一次完整的运行数据。另外，在收集最终的数字之前，再检查一次!

不要重复使用相同的数据。确保每次运行使用不同的数据。例如，在数据中要有时间戳或其他独特的标识符（如图中的坐标和标签）。这是为了确保你确实在读取正确的数据，而不是一些陈旧的缓存内容，错误的块，等等。

对同一数据点使用连续和单独运行的组合。例如，对同一个点至少连续做两次（有助于识别不应该存在的缓存效应），在其他一些点之后再做两次（以识别不应该存在的缓存情况）。好好看看标准偏差。

颠倒测量的顺序。这有助于识别测量之间的干扰。这一点和上一点可以通过在两个方向上遍历一组数据点来实现。

不要只使用常规的步幅或二的幂。你可能在不知不觉中碰到了病态的情况。抛出一些随机点可能是个好主意。然而，不要只使用随机点，你可能会遗漏病态的案例。病理情况的良好候选者是 2^n、$2^n - 1$、$2^n + 1$。

当比较不同配置的测量结果时（这是你通常做的），确保你使用完全相同的点，不要只是在同一区间内比较图形。

当得到有趣的结果时，要检查你是否在用苹果与苹果进行比较。例如，要确保系统在你想比较的两次运行之间处于尽可能相同的状态。例如，我们曾经遇到过这样的情况：Linux 上的基准测试结果受到操作系统在物理内存中的分配位置的影响，这在连续的运行中是不同的（并且对物理地址缓存中的冲突失误有巨大的影响）。

2.3 统计数据

- 始终做几次运行，并检查标准偏差。注意不正常的差异。在我们所做的那种测量中，标准偏差通常应小于 0.1%。如果你看到 >1%，就应该敲响警钟。

- 在某些情况下，忽略最高点或最低点是合理的（但这确实应该在经过适当的统计轮廓检测程序后才能做到），或者只看点的底线。然而，只有在你真的知道自己在做什么的情况下，才可以有选择地使用这种数据，并在你的论文/报告中明确说明。

2.4 时序

使用大量的迭代，以改善统计，消除时钟的颗粒性。

运行足够的热身迭代，这些迭代是不需要计时的。

将你想计时的东西隔离到一个函数中（如果你在为系统调用计时，已经做了）。

消除循环的开销（不要仅仅依靠它的小，要消除它）。最可靠的方法是运行两个版本的基准，除了用 noop 代替实际调用（函数或系统调用）外，其他都一样。在没有任何编译器优化的情况下运行循环（这就是为什么要把你想计时的东西放在一个函数中，这很重要，但这可能需要你单独处理函数的开销）。

对上面的 syscall 循环进行静态分析，并验证计时数字是否与你的预测相符。

使用适当的统计数据，即使在最后的论文中没有使用，检查方差也是一个重要的理智检查。